Naval Research Laboratory

Washington, DC 20375-5000

NRL Report 9131

A Method for Automatically Translating Trace Specifications into Prolog

C. A. MEADOWS

Computer Science and Systems Branch
Information Technology Division

September 30, 1988

ADA200993

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE — *Form Approved OMB No. 0704-0188*

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NRL Report 9131	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Research Laboratory — 6b. OFFICE SYMBOL *(If applicable)* Code 5593	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS *(City, State, and ZIP Code)* Washington, DC 20375-5000	7b. ADDRESS *(City, State, and ZIP Code)*
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Office of Naval Research — 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* Arlington, VA 22217-5000	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11. TITLE *(Include Security Classification)*
A Method for Automatically Translating Trace Specifications into Prolog

12. PERSONAL AUTHOR(S)
Meadows, C.A.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Interim	FROM 10/85 TO 10/87	1988 September 30	41

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Automatic implementation; Prolog [illegible]; Formal specification; Logic programming; [illegible] Rapid prototyping;

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The trace method of software specification provides unambiguous, nonprocedural program specifications that are subject to rigorous proof techniques. The nonprocedural property makes it easier for a programmer to develop an implementation without being overly influenced by the form the specification takes. However, this same nonprocedural property often makes it difficult for programmers to interpret trace specifications or for users to predict the results of the specifications they design. This disadvantage can be overcome by a rapid prototyping system that enables programmers and specification writers to immediately check the exact requirements of a given specification. Such a prototyping system should be able to take a given trace specification and automatically translate it into executable code that satisfies the requirements of the specification. In this report we present an algorithm for translating specifications into Prolog and a method for formulating specifications so that the algorithm always produces complete programs. Therefore a program thus produced will eventually yield an answer to any query concerning the value or legality of any given trace or the equivalence of any two given traces. We also provide a proof that this property holds. [illegible] programming

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Catherine A. Meadows	22b. TELEPHONE *(Include Area Code)* (202) 767-3490 — 22c. OFFICE SYMBOL Code 5593

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603

CONTENTS

A METHOD FOR AUTOMATICALLY TRANSLATING TRACE SPECIFICATIONS INTO PROLOG

1. INTRODUCTION

The trace method of software specification [1,2] provides unambiguous, nonprocedural program specifications that are subject to rigorous proof techniques. The nonprocedural property makes it easier for a programmer to develop an implementation without being overly influenced by the form the specification takes. However, this same nonprocedural property often makes it difficult for programmers to interpret trace specifications or for users to predict the results of the specifications they design.

This disadvantage can be overcome by a rapid prototyping system [3] that enables programmers and specification writers to immediately check the exact requirements of a given specification. Such a prototyping system should be able to take a given trace specification and automatically translate it into executable code that satisfies the requirements of the specification. A user can then run the resulting program and determine whether or not the specification actually requires what the user had in mind.

In their report *Executing Trace Specifications Using Prolog*, McLean, Weiss, and Landwehr [4] discuss several possible target languages for such an automatic translator. They decide on Prolog [5] as the most likely candidate, given the nonprocedural nature of its syntax. However, Prolog's dependence on depth-first search leads in many cases to logically incomplete programs, that is, programs that fail to return an answer to a query even though one exists. McLean et al. address this problem in their report and suggest that it can be solved by formulating the trace specification in such a way that the Prolog translation is always complete. They also present some suggestions for such formulations.

In this report we present a method of formulating trace specifications based on the work of Hoffman [6,7] and an algorithm for translating such specifications into complete and correct Prolog code. This algorithm has been implemented in the string manipulation language Icon [8]. We also present a proof that the algorithm always produces complete programs, in the sense that a program thus produced will eventually yield an answer to any query concerning the value or legality of any given trace or the equivalence of any two given traces.

2. DESCRIPTION OF THE TRACE SPECIFICATION METHOD

The trace specification method describes the behavior of a software module in terms of a sequence or *trace* of procedure and function calls and return values from these calls. The designer of a trace specification must supply three things. First, he or she must describe the procedure and function calls in terms of their names, parameter types, and return value types, if any. These are given in sentences of the form

(1) name: param_type,. . .,param_type -> return_value.

Manuscript approved April 4, 1988.

Second, the designer must specify which traces are legal, that is, which sequences of procedure and function calls the module will accept. This is done by writing sentences of the form

(2) Conditions $\rightarrow$ L(T).

Finally, the designer must describe the output of a legal trace ending in a function call by supplying sentences of the form

(3) Conditions $\rightarrow$ V(T) = value.

To make the designer's task easier, a notion of equivalence is also provided. Two traces are equivalent if they always agree on legality and return value with respect to future program behavior. Thus the designer can also supply sentences of the form

(4) Conditions $\rightarrow T_1 \equiv T_2$.

Two or more traces are concatenated by the use of the dot (.). That is, trace T_1 followed by T_2 is written as $T_1.T_2$.

We give an example of a specification for a table lookup program below:*

Syntax:

Insert: entry
Delete:
Left:
Right:
Current: --> entry

Legality:

(1) L(T) $\rightarrow$ L(T.insert(a))
(2) L(T) $\rightarrow$ L(T.Insert(a).Left)

Equivalence:

(3) T.Insert(a).Current $\equiv$ T.Insert(a)
(4) T.Insert(a).Delete $\equiv$ T
(5) T.Insert(a).Left.Right $\equiv$ T.Insert(a)
(6) T.Insert(a1).Left.Insert(a2) $\equiv$ T.Insert(a2).Insert(a1).Left

Value:

(7) L(T) $\rightarrow$ V(T.Insert(a).Current) = a.

One determines the value of a legal trace T.Current by manipulating T by the use of Axioms (3) through (7) until a trace ending in an insert call is obtained. For example,

Insert(1).Left.Insert(2).Right.Current $\equiv$ Insert(2).Insert(1).Left.Right.Current (by Axiom (6))
Insert(2).Insert(1).Left.Right.Current $\equiv$ Insert(2).Insert(1).Current (by Axiom (5))
V(Insert(2).Insert(1).Current) = 1 (by Axiom (7)).

*This specification is due to Jonathan Hilliar.

3. DESCRIPTION OF PROLOG

A line of code in Prolog is either a clause of the form

$$P(t_1,t_2,\ldots,t_n),$$

where P is an n-place predicate and each t_i is a term, or it is of the form

$$G :- F_1,F_2,\ldots,F_m$$

(that is, G if F_1 & F_2 & ... & F_m), where G is a predicate and the F_i's are predicates or negations of predicates. The first kind of clauses are known as facts, and the second kind are known as rules.

Since we are dealing with lists of procedure and function calls, it is useful to know how to represent lists in Prolog. A list of elements a_1 through a_n in Prolog is written as $[a_1,\ldots,a_n]$. A list can also be written as [X|Y], where X is the first element, or *head* of the list, and Y is the rest, or *tail* of the list. The empty list is represented by [].

A Prolog program works in the following way. A user queries a Prolog program by presenting it a goal in the form of a predicate $P(t_1,\ldots,t_n)$ or the negation of such a predicate, or by giving a sequence of such goals. Prolog tries to satisfy the goal $P(t_1,\ldots,t_n)$ by querying each fact and each rule concerning P until it finds one that P satisfies. If it finds such a rule, it returns the answer "yes." Otherwise it returns the answer "no." If some of the t_i's are variables that must be set to constants in order for a rule or fact to be satisfied, Prolog also returns those constants. If queried repeatedly (by a user typing ";"), Prolog returns all the constants or variables it can find that make P true.

Prolog checks whether or not a goal G satisfies a rule of the form

$$G :- F_1,F_2,\ldots,F_n$$

in the following way. It attempts to satisfy each goal F_i proceeding from left to right. If it satisfies F_i and cannot satisfy F_{i+1}, it backtracks to F_i and tries to satisfy it some other way. It continues doing this until it either satisfies F_{i+1} or until it runs out of ways of satisfying F_i.

For example, consider the following Prolog program designed to tell us whether or not a given list has a number greater than 5 as a member:

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).

ok(L) :- member(X,L), X > 5.
```

Suppose that Prolog is given the goal ok([2,1,6,3]). It first attempts to satisfy the goal member(X,[2,1,6,3]). Invoking the first rule, it obtains X = 2. It then attempts to satisfy the goal 2 > 5 and fails. It then backtracks to the goal member(X,[2,1,6,3]). It invokes the second rule and so now must satisfy the goal member(X,[1,6,3]). It does this by using the first rule and returns X = 1. It now attempts to satisfy the goal 1 > 5 and fails. It backtracks again to the goal member(X,[2,1,6,3]). It invokes the second rule and now must once again satisfy the goal member(X,[1,6,3]). It has already exhausted all ways of using the first rule to do this and so invokes the second rule. Now it must satisfy the goal member(X,[6,3]). It uses the first rule, and returns X = 6. It attempts to satisfy the goal 6 > 5 and succeeds.

It is clear that such a depth-first method of search necessarily allows incomplete programs, that is, programs that fail to return answers to questions even though the answers exist. One source of incompleteness lies in recursively defined predicates. For example, suppose that Prolog is given a definition of the form

```
enemy(dog,cat).
enemy(A,B) :- enemy(B,A).
```

If presented with the goal enemy(cat,dog), Prolog uses the rule and the fact given it to produce the answer yes. However, suppose that it is given the goal enemy(cat,horse). It then uses the rule and attempts to satisfy the goal enemy(horse,cat). Since that goal is not satisfied by the fact enemy(dog,cat), it uses the rule again and attempts to satisfy the goal enemy(cat,horse) and so on. Thus, although the answer to the query "enemy(cat,horse)" is "no" (since Prolog cannot show that the answer is yes), Prolog cannot return any answer.

Another source of incompleteness lies in the way Prolog backtracks to a goal upon failure of the goal following it and attempts to find all ways of satisfying the first goal until the second goal succeeds. If the first goal can be satisfied in an infinite number of ways, none of which satisfy the second goal, then Prolog again does not return an answer. For example, consider the following Prolog program defining a predicate called "append" that appends one list to another to form a third.

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

If we present the query append(X,[1,2,3],Y) to Prolog and repeat it, we get the infinite sequence of answers

```
X = []        Y = [1,2,3]
X = [V1]      Y = [V1,1,2,3]
X = [V2,V1]   Y = [V2,V1,1,2,3]
etc.,         ...
```

Thus, if we define the predicate sublist(H,L) by

```
sublist(H,L) :-
     append(X,H,T),
     append(T,S,L).
```

we never get an answer to the query sublist([1,2,3],[4,5,6]), although, since Prolog is unable to come up with a positive answer to the query, the answer is again no.

In the next section we discuss the bearing the incompleteness of Prolog has on the automatic translation of trace specifications.

4. SOME PROBLEMS WITH INCOMPLETENESS IN PROLOG TRANSLATIONS OF TRACE SPECIFICATIONS

In this section we consider the problems that arise when we attempt to guarantee completeness of Prolog translations of trace specifications.

Consider the table lookup specification presented in Section 2. It is an easy enough task to translate each line of that specification into a line of Prolog code, as we see in the example below.*

```
(1) l(T) :- append([[insert,A]],S,T), l(S).
(2) l(T) :- append([left,[insert,A]],S,T), l(S).
(3) equiv(T,S) :- append([current,[insert,A]],R,T),
        append([[insert,A]],R,S).
(4) equiv(T,S) :- append([delete,[insert,A]],S,T).
(5) equiv(T,S) :- append([right,left,[insert,A]],R,T),
        append([[insert,A]],R,S).
(6) equiv(T,S) :- append([[insert,A1],left,[insert,A2]],R,T),
        append([[left,[insert,A2],[insert,A1]],R,S).
(7) v(T,A) :- append([current,[insert,A]],S,T), l(T).
```

We also need code describing general properties of legality, equivalence, and value.

```
 (8) l([]).
 (9) l(S) :- append(R,S,T), l(T).
(10) l(T) :- equiv(T,S), l(S).
(11) equiv(S,T) :- append(R1,R,S), append(R2,R,T), equiv(R1,R2).
(12) equiv(S,T) :- equiv(T,S).
(13) equiv(S,T) :- equiv(S,R), equiv(R,T).
(14) equiv(T,T).
(15) v(T,A) :- equiv(T,S), v(S,A).
```

However, numerous problems arise when we try to use the program thus generated. Some of the problems arising from the use of the straightforward definition of equivalence have already been described. Other problems arise from the use of Axiom (9), which says that if a trace S.R is legal, then so is the trace S. For example, suppose that the program is given the illegal trace [left] and is asked to rule on its legality. Axioms (1), (2), and (8) obviously do not apply, so it proceeds to Axiom (9) and attempts to build a legal trace S such that T = S.R. The first solutions it comes up with are T = [left] and R = []. It then attempts to satisfy the other axioms, fails, and once again tries to satisfy Axiom (9). Thus it has entered an infinite loop.

This problem is relatively easy to correct; Axiom (9) can be omitted if the specification writer is careful to write specifications so that any trace that is a prefix of a legal trace can be shown to be legal without the use of Axiom (9). More details on how to do this are given in the next section. The problems arising from the axioms defining the general notion of equivalence (Axioms (12) through (14)) may be avoided by omitting these axioms and reinterpreting Axioms (3) through (6) as *reduction rules*, that is, rules for reducing complicated expressions to simpler, equivalent expressions. Reduction rules may be obtained by putting an ordering on trace expressions and interpreting an equivalence axiom A ≡ B as A --> B if B precedes A in the order or as B --> A if A precedes B. One then shows that two traces are equivalent by reducing them to the same irreducible trace. One determines the value of a trace by reducing it to a trace for which a value is defined.

Our use of reduction rules deviates slightly from the usual. In most reduction systems, a rule can be used to reduce any part of an expression. Thus a rule such as ab --> c may be used to show that dabf --> dcf. In our use of reduction rules, we only allow rules to be used to reduce prefixes of expressions. Thus a rule such as

*Note that the order of the procedure calls in the traces in this program is reversed. This is done for the sake of efficiency in the Prolog program.

l(T) => push(a).T.firstpop --> T

could be used to reduce

push(apple).push(orange).firstpop.push(kumquat)

to

push(orange).push(kumquat)

since push(apple).push(orange).firstpop reduces to push(orange), but could not be used to reduce the trace to

push(apple).push(kumquat)

even though push(orange).firstpop considered as a trace in itself reduces to the empty trace.

Replacing equivalence axioms by reduction rules does not in itself guarantee completeness, however, since it is possible that an expression may not reduce to an irreducible expression in a finite number of steps. Moreover, the use of reduction rules in place of equivalence axioms may also give rise to incorrectness, since it is possible that two equivalent traces may reduce to two distinct irreducible traces or that a trace T for which V(T.C) is explicitly defined may reduce to a trace T′ for which V(T′.C) is not explicitly defined.

Incompleteness arising from nontermination of reduction systems may be avoided by basing the reduction rules on a well-founded partial ordering of expressions, that is, one in which there is no infinite chain of expressions

$$X_1 > X_2 > \ldots > X_n > \ldots$$

In such a case, any expression must reduce to an irreducible expression in a finite number of steps, since otherwise it would be preceded by an infinite number of expressions.

The problem of guaranteeing the correctness of a reduction system is trickier. Consider our table lookup specification, for example. We may reinterpret Axioms (3) through (6) as

(3) T.insert(a).current --> T.insert(a)
(4) T.insert(a).delete --> T
(5) T.insert(a).left.right --> T.insert(a)
(6) T.insert(a1).insert(a2).left --> T.insert(a2).left.insert(a1).

It can easily be shown that this reduction system terminates. (We leave it as an exercise to the reader to find the well-founded partial order with which it is consistent.) But suppose that we want to determine the value of the trace

Insert(apple).Left.Insert(orange).Right.Current.

We can no longer use Axiom (6) to show that the trace is equivalent to

Insert(orange).Insert(apple).Left.Right.Current and hence equivalent to

Insert(orange).Insert(apple).Current

with value "apple." Moreover, we cannot solve our problem by replacing Axiom (6) with the axiom

(6′) T.Insert(a2).Left.Inscrt(a1) --> T.Insert(a1).Insert(a2).Left

since then we would no longer be able to find the value of the trace

Insert(apple).Insert(orange).Left.Current.

One possible solution to this problem is to add more axioms to the specification. Thus, for example, we could insert the axiom

(6.1a) T.insert(a1).left.insert(a2).right --> T.insert(a2).insert(a1).

But this would give us no help in evaluating

v(insert(apple).left.insert(orange).insert(kumquat).current),

and we would have to insert a new axiom to cover that case. It is easy to see that eventually we would wind up with an axiom for each occurrence of a left followed by N inserts. Since this would leave us with an infinite number or axioms, it is clearly undesirable. If we substitute axiom (6′) for axiom (6), we run into the same problem although in this case we have to add an infinite number of value axioms instead of an infinite number of reduction rules.

Our solution to the problem of incorrect reduction systems arising from correct specifications is to design specifications that can be guaranteed to give rise to correct reduction systems. In the next section we show how a heuristic developed by Daniel Hoffman [6] in order to make specifications easier to write and understand can be used to assist in the design of specifications that can be interpreted as correct reduction systems.

Another problem arises from the use of Prolog predicates in an inappropriate context. Consider, for example, the use of "append" in the following axiom:

```
equiv(A,B) :- append([up],L1,T), append(H,T,A), append([down],L1,L2),
              append(H,L2,B), val(B,5).
```

Suppose that we are using this axiom to find a trace B that is equivalent to A = [out,push(w)]. Prolog first attempts to satisfy the query append([up],L1,T). This it can do successfully, but an infinite number of traces L1 and T satisfy this condition. Thus, when the next condition, append(H,T,A), fails, Prolog will be forced to backtrack an infinite number of times, thus causing an infinite loop. This is not the case if we reverse the order of the first two conditions, since only a finite number of traces H and T satisfy append(H,T,A). However, we do *not* wish to reverse the order of the next two conditions. For suppose that we had done so, and that we wished to use the axiom to find a trace B equivalent to A = [up,out]. The first two conditions would be satisfied successfully, instantiating H. However, B and L2 would still be variables, and thus there would be an infinite number of traces satisfying append(H,L2,B). If there were no such B with value 5, the program would again be forced to backtrack an infinite number of times. Thus a traces-to-Prolog translator must be designed to take into account the context in which a condition will appear, that is, what positioning of a condition will allow it to be satisfied an infinite number of times and what positioning will allow it to be satisfied only a finite number of times.

5. HOFFMAN'S HEURISTIC FOR TRACE SPECIFICATIONS

In this section we present the techniques introduced by Hoffman [6] that we plan to use in designing specifications that can be easily translated into Prolog.

Hoffman's first innovation is to introduce functions that allow one to make global statements about traces. The four functions our translator recognizes as of now are

(1) length(T) = number of procedure calls in trace T
(2) count(c,T) = number of time call c appears in trace T
(3) all(c,T) = "true" if T consists entirely of calls named c
(4) nth(T,n) = the nth call appearing in trace T.

Count(c,T) can actually be used in two ways. It can be used to ask how many calls with a given name appear in T. Thus, count(insert, insert(1).insert(2).delete) = 2. Or it can be used to ask how many times a particular call appears in T. Thus, count(insert(1), insert(1).insert(2).insert(1)) = 2.

The addition of these functions has no effect on the foundations underlying trace specifications, but the new functions do allow us to make more general statements about traces than would otherwise be possible. For example, the infinite set of axioms concerning Inserts, Left, and Right from Section 4 can now be replaced by the single axiom

L(T2.Insert(a).Left.T1.Right) &
all(Insert,T1) &
length(T1) > 0
  → T2.Insert(a).Left.T1.Right ≡ T2.T1.Insert(a).

Thus the introduction of functions allowing us to make global statements about traces makes it easier to write specifications that can be interpreted as correct reduction systems. However, they do not give us a general method of writing such specifications. That is provided by the technique that follows.

Suppose that a specification writer has defined legality and value for a set of traces. Then it is possible to divide the set of legal traces into equivalence classes. Choose a representative of each class such that each prefix of a representative is also a representative. We say a trace T is in *normal form* if it is a representative of an equivalence class of legal traces.

The conclusions of our trace specifications can now take three possible forms. A conclusion can be of the form isnf(T) (T is in normal form), T.C ≡ S, where T and S are in normal form, C is a single procedure call, or v(T.C) = A, where T is in normal form and C is a single procedure call.

One proves assertions concerning value and legality by use of the following three axioms:

(1) isnf(e). (where "e" stands for the empty trace)
(2) T = S.C.R & isnf(S) & length(C) = 1 & not(isnf(S.C)) & S.C ≡ Q
    & l(Q.R) => l(T).
(3) T = S.C.R & isnf(S) & length(C) = 1 & not(isnf(S.C)) & S.C ≡ Q
    & length(D) = 1 & v(Q.R.D) = X => v(T.D) = X.

We give an example of the table lookup specification as written by Hoffman to conform to his heuristics:

all(insert,T1) & all(goleft,T2) & length(T2) $\leq$ length(T1) $\rightarrow$ isnf(T1.T2)
isnf(T) $\rightarrow$ T.current $\equiv$ T
isnf(T.goleft) $\rightarrow$ T.goleft.goright $\equiv$ T
isnf(T1.insert(a).T2.T3) & all(insert,T2) & all(goleft,T3) & length(T2) = length(T3)
    $\rightarrow$ T1.insert(a).T1.T2.delete $\equiv$ T1.T2.T3
isnf(T1.T2.T3) & all(insert,T2) & all(goleft,T3) & length(T2) = length(T3)
    $\rightarrow$ T1.T2.T3.insert(a) $\equiv$ T1.insert(a).T2.T3
isnf(T1.insert(a).T2.T3) & all(insert,T2) & all(goleft,T3) & length(T2) = length(T3)
    $\rightarrow$ V(T1.insert(a).T2.T3.current) = a.

Hoffman also includes axioms about legality. These axioms are not necessary, however, since it is possible to prove a trace legal or illegal by proving that it is equivalent to a trace in normal form or that it is not equivalent to any trace in normal form.

6. A GRAMMAR FOR TRANSLATABLE TRACE SPECIFICATIONS

Recasting trace specifications as reduction systems, although it removes one source of infinite looping, does not remove infinite loops arising from recursive definitions of trace predicates and functions. Consider, for example, the Hoffman-legal assertion:

isnf(T) & V(T.T.current) = X $\rightarrow$ V(T.current) = X/2.

Prolog will discover the value of a trace T.current by using the rule to find V(T.T.current), then to find V(T.T.T.T.current), and so on. We can prevent such looping by requiring the specification writer to put a well-founded order on trace expressions and to define trace assertions such that whenever a predicate A(T_1,...,T_k) is used in the proof of A(S_1,...,S_k), then the T_i's precede the T_k's in the partial order.

Another source of infinite looping lies, as we have said before, in conditions that can be satisfied in an infinite number of ways, and thus cause an infinite amount of backtracking. Consider, for example, the assertion

isnf(T) & v(Q.current) = 5 & T = S.Q.R $\rightarrow$ V(T.current) = 5.

If there are an infinite number of traces Q such that v(Q.current) = 5, but T = S.Q.R for no such Q, then Prolog, even if it is able to generate all such Q, is caught in an infinite loop, since it backtracks to V(Q.current) = 5 and tries to satisfy it in a different way each time it fails on T = S.Q.R. This problem can be solved by reordering the conditions:

isnf(T) & T = S.Q.R & V(Q.current) = 5 $\rightarrow$ V(T.current) = 5.

Only a finite number of Q, S, and R exist such that T = S.Q.R, so Prolog does not get caught in an infinite loop when it is presented with a trace T and asked to find its value.

In this section we provide a grammar for translatable specifications. This grammar is a version of the grammar developed by McLean [2] with some restrictions. First we describe the vocabulary of our trace specification language, and then we show how it differs from McLean's.

(1) **Trace Variables.** B,C,...,X,Y with subscripts if necessary. The letters A and Z are reserved for the translator.

(2) **Empty Trace.** Denoted by e.

(3) **Procedure Name.** Any finite character string beginning with a lowercase letter is a procedure name. When composed with the appropriate parameters, it forms a procedure call.

(4) **Trace Predicates.** We allow four trace predicates: the unary predicates isnf(T) and all(c,T) and the binary predicates T1 == T2 and T1 === T2. This last stands for equality of traces; the nonstandard equality notation is necessary since the translator must follow a special procedure when two traces (as opposed to two terms of some other domain type) are set equal to each other.*

(5) **Trace Functions.** The dot (.), V, length, count, and nth are the trace functions.

(6) **Domain Names.** The name of any domain. As in McLean, we are primarily interested in parameter domains for procedure calls.

(7) **Domain Constants.** Domain constants are numerals, integers, or character strings *not* beginning with capital letters.

(8) **Arithmetic Operations.** The arithmetic operations take the place of the domain functions defined in McLean. They are the only domain functions we allow. They are +, -, *, /, and ** (for exponentiation).

(9) **Domain Relations.** The permissible domain relations are =, and >, <, =<, >=, for domains for which they are defined.

(10) **Domain variables.** B,C,...,X,Y with subscripts if necessary. Domain variables are distinguishable from trace variables by context.

(11) **Connectives.** The connectives are & (for and), | (for or), if, then, and not.

(12) **Parentheses.** (and).

The rules of formation for trace specifications are as follows.

Domain Lists:

domain list →
  domain name |
  domain list domain name

Syntax Sentences:

syntax sentence →
  procedure name: |
  procedure name: domain list |
  procedure name: => domain name |
  procedure name: domain list => domain name

*Note that L(T) is no longer a language primitive. We can introduce it as an abbreviation L(T) = "there exists S such that isnf(S) & T ≡ S".

Domain Elements:

domain element →
domain constant |
domain variable

Variables:

variable →
domain variable |
trace variable

Argument Lists:

argument list →
domain element |
argument list, domain element

Procedure calls:

procedure call →
procedure name |
procedure name(nonempty argument list)

Trace Expressions:

trace expression →
empty trace |
trace variable |
procedure call |
trace expression.trace expression

Terms:

term →
domain element |
trace expression |
V(trace expression) |
nth(integer element,trace expression) |
count(procedure name,trace expression) |
count(procedure call,trace expression) |
length(trace expression)

As in McLean, terms and arithmetic expressions can have various *types*. Domain elements inherit their type from the type of the domain to which they belong. Trace expressions are of type *trace expression*. Terms of the form *V(trace expression.procedure call)* are of the same type as the type of the domain element returned by the procedure call.

Arithmetic Expressions

> **arithmetic expression →**
> **integer or real constant |**
> **domain variable |**
> **term of type integer or real |**
> **(arithmetic expression) + (arithmetic expression) |**
> **(arithmetic expression) * (arithmetic expression) |**
> **(arithmetic expression) - (arithmetic expression) |**
> **(arithmetic expression) / (arithmetic expression) |**
> **(arithmetic expression) ** (arithmetic expression)**

Arithmetic expressions are of type integer or real.

Predicate Expressions:

> **predicate expression →**
> **isnf(trace expression) |**
> **trace expression == *trace expression* |**
> **trace expression === trace expression |**
> **term = term |**
> **arithmetic expression domain relation arithmetic expression**

Conditions:

> **condition →**
> **predicate expression |**
> **not(condition) |**
> **if (condition) then (condition) |**
> **(condition) "|" (condition) |**
> **(condition) & (condition)**

The quotation marks around the "|" sign are to distinguish it from the "|" sign used to build the grammar and do not appear in the actual specification. A *condition such as not* (X = Y) may also be written as X != Y.

A condition of the form "not (condition)" or "if (condition) then (condition)" is called a *negated* condition. A condition of the form "((condition)|(condition)|...|(condition))" is called an *or-condition*. A condition of the form "((condition)&(condition)&...&(condition))" is called an *and-condition*. Parentheses around a condition used to build an or-condition are necessary if and only if that condition is an and-condition. Likewise, parentheses around a condition used to build an and-condition are necessary if and only if that condition is an or-condition.

Assertions:

> **normal-form assertion →**
> **isnf(trace expression) |**
> **condition => isnf(trace expression)**

> **value assertion →**
> **V(trace expression) = term |**
> **condition => V(trace expression) = term**

equivalence assertion →
  trace expression == trace expression |
  condition => trace expression == trace expression

assertion →
    normal-form assertion |
    value assertion |
    equivalence assertion

The *conclusion* of an assertion is defined to be the assertion itself if no conditions are present; otherwise it is defined to be the the part of the assertion following the "=>" symbol.

Syntax Section:

syntax section →
    syntax sentence |
    syntax section syntax sentence

Normal Form Section:

normal form section →
    normal-form assertion |
    normal-form section normal-form assertion

Value Section:

value section →
    value assertion |
    value section value assertion

Equivalence Section

equivalence section →
    equivalence assertion |
    equivalence section equivalence assertion

Semantics Section:

semantics section →
    normal form section value section equivalence section

Trace specification:

trace specification →
    syntax section semantics section

Specification assertions are defined to be *admissible* assertions. Admissibility is defined below.

We begin by defining admissible assertions in terms of assertions containing only the "&" connective.

Table 1 — Table of Occurrence Types

Predicate Expression	ω = occurrence of σ as a variable in	type of ω
$\phi == \psi$	ϕ ψ	"in" "out"
isnf(ϕ)	ϕ	"in"
all(π,ϕ)	π ϕ	"out" "in"
$\phi === \psi$	ϕ ψ	"in" "out"
$\alpha = \beta$ α and β single variables or procedure calls with variable arguments	α β	"out" "in"
$\alpha = \beta$ $\beta = \alpha$ α single variable or procedure call with variable arguments β not single variable or procedure call with variable arguments	α	"out"
Γ R count(π,ϕ) R = "=," "<," ">," "=," or "="	π ϕ	"out" "in"
Γ R v(ϕ) R = "=," "<," ">," "=" or "="	ϕ	"in"
Γ = nth(κ,ϕ) nth(κ,ϕ) = Γ	κ ϕ	"out" "in"
Γ R f(α1,. . .,αn) f(α1,. . .,αn) R Γ R = "=," "<," ">," "=" or "="	αi where αi a single variable κ in αi = count(κ,ϕ) ϕ in αi = count(κ,ϕ) ϕ in αi = length(ϕ)	"in" "out" "in" "in"
$\alpha > \Gamma$ $\alpha < \Gamma$ $\alpha \geq \Gamma$ $\alpha \leq \Gamma$ $\Gamma > \alpha$ $\Gamma < \alpha$ $\Gamma \geq \alpha$ $\Gamma \leq \alpha$	α where α is a single variable	"in"

Definition 6.1: We say that an assertion is *simple* if its condition is either empty, consists of a single atomic condition, or consists of atomic conditions bound together by the "&" connective.

To proceed further we need to define an occurrence of a variable or expression in an assertion.

Definition 6.2: Let σ be a substring of a character string S. An *occurrence* of σ in S is an ordered pair $\omega = (k_\omega, \sigma)$, where k_ω is an integer such that, for $1 \leq i \leq \text{length}(\sigma)$, the $k_\omega + i - 1$th character of S is the ith character of σ. We say that an occurrence ω' *precedes* ω if $k_{\omega'} < k_\omega$. We say that ω is an occurrence of σ in τ in S (more briefly, an occurrence of ω in τ, where confusion may be avoided) if there exists an occurrence ω' of τ in S such that $k_{\omega'} \leq k_\omega$ and $k_\omega + \text{length}(\sigma) \leq k_{\omega'} + \text{length}(\tau)$. We say that ω is an occurrence of *σ as a variable* in S (or as a constant, predicate, etc.) if σ is to be interpreted as standing for a variable (or for a constant, predicate, etc.) at the point at which it occurs.

We now assign types to occurrences of character strings as variables in predicate expressions occurring in assertions. An occurrence may be one of two types: "in"or "out." We motivate the definition of "in" and "out" as follows. If a variable must be instantiated before an expression is evaluated, its occurrence is of type "in" (for input); if it will be instantiated after the expression is evaluated, its occurrence is of type "out" (for output).

The various occurrence types are defined in Table 1.

In the example

$$W = \text{count}(N,T) + \text{count}(\text{goleft},T.S)$$

we see that the type of the occurrence of W in the left-hand side of the equation is "out," the occurrence of N in the first argument of the first "count" is of type "out," the occurrences of T in the second argument of the first "count" and in the second argument of the second "count" are both of type "in," and the occurrence of S in the second argument of the second "count" is of type "in."

We are now ready to define the context-sensitive portion of the grammar. This definition is given in terms of "admissible" assertions. We begin by defining admissibility of simple assertions and then use that definition of admissibility to define admissible general assertions. Admissibility is defined in two parts: variable admissibility, which provides a solution to the subgoal ordering problem, and trace admissibility, which provides a solution to the recursive looping problem.

Definition 6.3: We say that a simple assertion is variable admissible if

(1) whenever there is an occurrence of a variable in an "out" argument of the conclusion, then there is an occurrence of that same variable in either an "in" argument of the conclusion or in an "out" argument of a condition and

(2) whenever there is an occurrence ω of a variable in an "in" argument of a condition, then either there is an occurrence of that same variable in an "in" argument of the conclusion or there is an occurrence ω' preceding ω of that variable in an "out" argument of a condition.

The second condition, trace admissibility, is more complicated.

Definition 6.4: We say that a partial order $<$ on trace expressions is a *trace order* if

(1) $<$ is well founded;

(2) $A < B$ implies that $A\tau < B\tau$, where τ is any substitution for the variables in A and B, and;

(3) $A < B$ implies that $A.C < B.C$ for any trace expression C.

Definition 6.5: Let **S** and **T** be two sets of trace expressions, and let $<$ be a trace order. We say that **S** $<^*$ **T** if there exists an expression $S \in$ **S** and an expression $T \in$ **T** such that $S < T$. We say that **S** $\equiv^*$ **T** if S = T and that **S** $\leq^*$ **T** if **S** $<^*$ **T** or **S** $\equiv$ **T**.

Note that $<^*$ is *not* a partial order; in particular, it is not transitive. Moreover, it is possible to have **S** $\equiv$ **T** and **S** $<^*$ **T** be true for at the same time.

Definition 6.6: Let ω be an occurrence in an assertion. We define $G(\omega)$ to be the graph with node set $N(\omega)$ consisting of all trace expressions in the assertion and edge set consisting of all pairs (A,B) such that the equation A === B precedes ω. If S is a trace expression, we define $G^*(\omega,S)$ to be the connected component of $G(\omega)$ containing S; its node set is $N^*(\omega,S)$.

We are now ready to define trace admissibility.

Definition 6.7: Let $<$ be a trace ordering. We say that a simple assertion is trace admissible with respect to $<$ if

(1) whenever the conclusion is of the form "isnf(T)", then the condition contains no occurrences of X == Y or V(S), and whenever the conclusion is of the form X == Y, then the condition contains no occurrences of V(S);

(2) whenever ω is an occurrence of S as an "in" argument of type "trace" in a predicate in the condition of the same kind (isnf, V, ==) as the conclusion, then there is an "in" argument T of the conclusion such that then $N^*(\omega,S) <^* N^*(\omega,T)$; and

(3) whenever the conclusion is of the form T == T′ and ω is the occurrence of the end of the assertion, then $N^*(\omega,T') <^* N^*(\omega,T)$.

Definition 6.8: Let $<$ be a trace ordering. A simple assertion is admissible with respect to $<$ if it is both trace admissible with respect to $<$ and variable admissible.

Definition 6.9: Let $<$ be a trace ordering. Let A, B, C, and D be (possibly empty) conditions. Let E be a conclusion.

(1) The assertion A & not(B) & C $\rightarrow$ E is admissible with respect to $<$ if and only if both

A & C $\rightarrow$ E
and
A & B & C $\rightarrow$ E

are admissible with respect to $<$.

(2) The assertion A & (if B then C) & D → E is admissible with respect to < if and only if the assertions

A & D → E;
A & B & D → E;
and
A & B & C & D → E

are admissible with respect to <.

(3) The assertion A & (B | C) & D → E is admissible with respect to < if and only if the assertions

A & B & D → E
and
A & C & D → E

are admissible with respect to <.

For example, the assertion

v(T) = X → R == T

is not variable admissible, since the only occurrence of T in the condition is of type "in," and there is no type "in" occurrence of T in the conclusion. On the other hand the assertion

T == S & v(S) = X → v(T) = X

is variable admissible. There are two occurrences of S in the condition. The first is of type = "out," and the second is of type "in." There is a type "in" occurrence of T in the condition, but also a type "in" occurrence of T in the conclusion. There is a type "out" occurrence of X in the conclusion, but also a type "out" occurrence of X in the condition. Likewise the assertion

(T == S | Q == S) & A = v(S) → v(T.Q) = A

is variable admissible. For example, there is an "in" occurrence of S in A =v(S), but there are "out" occurrences of S in both parts of the or-condition preceding A = v(S). The assertion

v(T) != X & X > 5 → v(T) = 3

is not variable admissible, even though there is an "out" occurrence of X preceding the "in" occurrence of X in X > 5, since the "out" occurrence appears in a negated condition not containing the "in" occurrence. On the other hand, the assertion

not(v(T) = X & X > 5) → v(T) = 3

is variable admissible.

The assertion

isnf(T) → isnf(T.S)

is not trace admissible for any trace order $<$. Suppose that it were. Then the definition of trace admissibility implies that $T.S < T$, and the definition of trace order implies that $T < T$ by substitution of the empty trace. This would imply the existence of an infinite chain $\ldots T < T < T$, contradicting the well-foundedness of $<$.

On the other hand, the assertion

$$S = T.push(a).pop \;\&\; isnf(T) \rightarrow isnf(S)$$

is trace admissible with respect to the ordering based on length of traces. To see that this is true, let ω be the occurrence of T as the argument of isnf(T). Then $N^*(\omega,T) = \{T\}$, and $N^*(\omega,S) = \{S, T.push(a).pop\}$, and we see that $N^*(\omega,T) <^* N(\omega,S)$, since $T < T.push(a).pop$ according to our ordering.

Definition 6.10: A trace specification is *admissible* if there exists a trace order $<$ such that all trace assertions in the semantics section are admissible with respect to $<$.

7. DESCRIPTION OF THE TRANSLATION ALGORITHM

In this section we describe the algorithm for translating trace specifications into Prolog. We construct a function "trans" that maps trace specification assertions to Prolog statements.

```
BEGIN

i:=0, j:= 0

Assertions:

(1) Normal form assertions

    IF $\phi = \alpha_1. \ldots .\alpha_n$ such that $\alpha_1$ through $\alpha_n$ are trace variables or procedure calls AND n > 1
    THEN

            i:=i+1
            trans(condition => isnf($\phi$)) :=
                isnf(Ai) :-
                        trans(condition),
                        listappend([trans($\alpha_n$),trans($\alpha_{n-1}$),...,trans($\alpha_1$)],$A_i$).
            trans(isnf($\phi$) :=
                insf(Ai) :-
                        listappend([trans($\alpha_n$),trans($\alpha_{n-1}$),...,trans($\alpha_1$)],$A_i$).

    ELSE

            trans(condition => isnf($\phi$)) :=
                isnf(trans($\phi$)) :- trans(condition).
            trans(isnf($\phi$)) :=
                isnf(trans($\phi$)).

(2) Value assertions

    IF $\phi = \alpha_1. \ldots .\alpha_n$ such that $\alpha_1$ through $\alpha_n$ are trace variables or procedure calls AND n > 1
    AND $\kappa$ is a function term or arithmetic expression THEN
```

```
        i:=i+1
        j:=j+1
        trans(condition => v($\phi$) = $\kappa$) :=
            v(Ai,Zj) :-
                    listappend([trans($\alpha_n$),trans($\alpha_{n-1}$),...,trans($\alpha_1$)],Ai),
                    trans(condition),
                    trans(Zj = $\kappa$).

        trans(v($\phi$) = $\kappa$) :=
            v(Ai,Zj) :-
                    listappend([trans($\alpha_n$),trans($\alpha_{n-1}$),...,trans($\alpha_1$)],Ai),
                    trans(Zj = $\kappa$).

ELSE IF $\phi$ = $\alpha_1$. ... .$\alpha_n$ such that $\alpha_1$ through $\alpha_n$ are trace variables or procedure calls AND n
> 1 THEN

        i:=i+1
        trans(condition => v($\phi$) = $\kappa$) :=
            v(Ai,trans($\kappa$)) :-
                    listappend([trans($\alpha_n$),trans($\alpha_{n-1}$),...,trans($\alpha_1$)],Ai),
                    trans(condition).

        trans(v($\phi$) = $\kappa$) :=
            v(Ai,trans($\kappa$)) :-
                    listappend([trans($\alpha_n$),trans($\alpha_{n-1}$),...,trans($\alpha_1$)],Ai).

ELSE IF $\kappa$ is a function term or arithmetic expression THEN

        j:=j+1
        trans(condition => v($\phi$) = $\kappa$) :=
            v(trans($\phi$),Zj) :-
                    trans(condition),
                    trans(Zj = $\kappa$).

        trans(v($\phi$) = $\kappa$) :=
            v(trans($\phi$),Zj) :-
                    trans(Zj = $\kappa$).

ELSE

        trans(condition => v($\phi$) = $x$) :=
            v(trans($\phi$),trans($x$)) :-
                    trans(condition).

        trans(v($\phi$) = $x$) :=
            v(trans($\phi$),trans($x$)).
```

(3) Equivalence assertions

IF $\phi = \alpha_1. \ldots .\alpha_n$ such that α_1 through α_n are trace variables or procedure calls AND $\psi = \beta_1. \ldots .\beta_m$ such that β_1 through β_m are trace variables or procedure calls AND n > 1 AND m > 1 THEN

```
        i:=i+2
        k:=i-1
        trans(condition => $\phi \equiv \psi$) :=
            equiv(Ak,Ai) :-
                    listappend([trans($\alpha_n$),trans($\alpha_{n-1}$),...,trans($\alpha_1$)],Ak),
                    trans(condition),
                    listappend([trans($\beta_m$),trans($\beta_{m-1}$),...,trans($\beta_1$)],Ai).

        trans($\phi \equiv \psi$) :=
            equiv(Ak,Ai) :-
                    listappend([trans($\alpha_n$),trans($\alpha_{n-1}$),...,trans($\alpha_1$)],Ak),
                    listappend([trans($\beta_m$),trans($\beta_{m-1}$),...,trans($\beta_1$)],Ai).

ELSE IF $\phi = \alpha_1. \dots .\alpha_n$ such that $\alpha_1$ through $\alpha_n$ are trace variables or procedure calls AND n
> 1 THEN

        i:=i+1
        trans(condition => $\phi \equiv \psi$) :=
            equiv(Ai,trans($\psi$)) :-
                    listappend([trans($\alpha_n$),trans($\alpha_{n-1}$),...,trans($\alpha_1$)],Ai),
                    trans(condition).

        trans($\phi \equiv \psi$) :=
            equiv(Ai,trans($\psi$)) :-
                    listappend([trans($\alpha_n$),trans($\alpha_{n-1}$),...,trans($\alpha_1$)],Ai).

ELSE IF $\psi = \beta_1. \dots .\beta_m$ such that $\beta_1$ through $\beta_m$ are trace variables or procedure calls AND
m > 1 THEN

        i:=i+1
        trans(condition => $\phi \equiv \psi$) :=
            equiv(trans($\phi$),Ai) :-
                    trans(condition),
                    listappend([trans($\beta_m$),$trans(f(\beta_{m-1})$,...,$trans(\beta_1)$],Ai).

        trans($\phi \equiv \psi$) :=
            equiv(trans($\phi$)),Ai) :-
                    listappend([trans($\beta_m$),$trans(f(\beta_{m-1})$,...,$trans(\beta_1)$],Ai).

ELSE

        trans(condition => $\phi \equiv \psi$) :=
            equiv(trans($\phi$),trans($\psi$)) :-
                    trans(condition).

        trans($\phi \equiv \psi$) :=
            equiv(trans($\phi$),trans($\psi$)).
```

Conditions:

trans(Φ & Ψ) := (trans(Φ)), (trans(Ψ))
trans(Φ | Ψ) := (trans(Φ)); (trans(Ψ))
trans(not(Φ)) := not((trans(Φ)))
trans(if Φ then Ψ) := not((trans(Φ),not((trans(Ψ))))).

Predicate Expressions:

(1) Relations involving arithmetic expressions

$f(\alpha_1,\ldots,\alpha_n)$ R $g(\beta_1,\ldots,\beta_m)$

where f and g stand for arithmetic expressions, α_1 through α_n are the terms of f, β_1 through β_n are the terms of g, and R is a relation.

```
IF there exists an s such that $\alpha_s$ is a function expression THEN
        j:=j+1
        trans(f($\alpha_1$,...,$\alpha_s$,...,$\alpha_n$) R g($\beta_1$,...,$\beta_m$)) :=
            trans(Zj = $\alpha_s$), trans(f($\alpha_1$,...,Zj,...,$\alpha_n$) R g($\beta_1$,...,$\beta_m$)).
ELSE IF there exists a t such that $\beta_t$ is a function expression THEN
        j:=j+1
        trans(f($\alpha_1$,...,$\alpha_n$) R g($\beta_1$,...$\beta_t$,...,$\beta_m$)) :=
            trans(Zj = $\beta_t$), trans(f($\alpha_1$,...,$\alpha_n$) R g($\beta_1$,...,Zj...,$\beta_m$)).
ELSE IF n > 1 THEN
        j:=j+1
        trans(f($\alpha_1$,...,a_n) R g($\beta_1$,...,$\beta_m$)) :=
            Zj is f($\alpha_1$,...,$\alpha_n$), trans(Zj R g($\beta_1$,...,$\beta_m$)).
ELSE IF m > 1 THEN
        j:=j+1
        trans($\alpha$ R g($\beta_1$,...,$\beta_m$)) :=
            Zj is g($\beta_1$,...,$\beta_m$), trans($\alpha$ R Zj).
ELSE
        trans($\alpha$ R $\beta$) := $\alpha$ R $\beta$.
```

(2) Equivalence of traces

trans(ϕ == ψ).

IF $\phi = \alpha_1. \ldots .\alpha_n$ where the α_i are trace variables of procedure calls AND $n > 1$ THEN

```
        i := i+1
        trans($\phi$ == $\psi$) :=
            listappend([trans($\alpha_n$),trans($\alpha_{n-1}$),...,trans($\alpha_1$)],Ai),
            trans(Ai == $\psi$).
```

ELSE IF $\psi = \beta_1. \ldots .\beta_m$ where the β_i are trace variables or procedure calls AND $m > 1$ THEN

```
        i:=i+1
        trans($\phi$ == $\psi$) :=
            trans($\phi$ == Ai),
            listappend([trans($\beta_m$),trans($\beta_{m-1}$),...,trans($\beta_1$)],Ai).
```

ELSE

```
        trans($\phi$ == $\psi$) := equiv(trans($\phi$),trans($\psi$)).
```

(3) Equality of trace expressions:

$$\phi === \psi$$

IF $\phi = \alpha_1. \ldots .\alpha_n$ where the α_i are trace variables or procedure calls AND $n > 1$ THEN

```
        i := i+1
        trans($\phi$ === $\psi$) :=
            listappend([trans($\alpha_n$),trans($\alpha_{n-1}$),...,trans($\alpha_1$)]),
            trans(Ai === $\psi$).
```

ELSE IF $\psi = \beta_1. \ldots .\beta_m$ where the β_i are trace variables or procedure calls AND $m > 1$ THEN

```
        i:=i+1
        trans($\phi$ === $\psi$) :=
            trans($\phi$ === Ai),
            listappend([trans($\beta_m$),trans($\beta_{m-1}$),...,trans($\beta_1$)]).
```

ELSE

```
        trans($\phi$ === $\psi$) := trans($\phi$) = trans($\psi$).
```

(4) Other trace predicates: isnf, all

$$\text{pred}(\alpha_1,\ldots,\alpha_n).$$

IF there exists an s such that α_s is a trace expression $\beta_1. \ldots .\beta_m$ where the β_i's are trace variables or procedure calls AND $m > 1$ THEN

```
        $i := i+1$
        $trans(pred(\alpha_1,\ldots,\alpha_s,\ldots,\alpha_n)) :=$
            $(listappend([trans(\beta_m),\ldots,trans(\beta_1)],Ai),$
            $trans(pred(\alpha_1,\ldots,Ai,\ldots,\alpha_n)).$

ELSE

        $trans(pred(\alpha_1,\ldots,\alpha_n)) :=$
            $pred(trans(\alpha_1),\ldots,trans(\alpha_n)).$
```

(5) Equations involving functions

$\phi(\alpha_1,\ldots,\alpha_n) = \psi$ (or $\psi = \phi(\alpha_1,\ldots\alpha_n)$) where ϕ = length, count, nth, or value, and ψ is not a function term or a complex arithmetic expression

IF there exists an s such that α_s is a trace expression $\alpha_s = \beta_1.\ldots.\beta_m$ where the β_t's are trace variables or procedure calls AND $m > 1$ THEN

```
        $i := i+1$
        $trans(\phi(\alpha_1,\ldots,\alpha_s,\ldots,\alpha_n) = \psi) :=$
            $(listappend([trans(\beta_m),\ldots,trans(\beta_1)],Ai),$
            $trans(\phi(\alpha_1,\ldots,Ai,\ldots,\alpha_n) = \psi)).$

ELSE

        $trans(\phi(\alpha_1,\ldots,\alpha_n) = \psi) :=$
            $\phi(trans(\alpha_1),\ldots,trans(\alpha_n),trans(\psi)).$
```

Trace Expressions:

```
IF $\phi$ is a trace variable THEN $trans(\phi) = \phi$
ELSE IF $\phi$ is a procedure call THEN $trans(\phi) := [trans(\phi)]$
ELSE IF $\phi = e$ THEN $trans(\phi) := []$.
```

Procedure Calls:

(1) Empty argument list

$trans(f) := [f]$

(2) Nonempty argument list

$trans(f(a_1,\ldots,a_n)) := [f,a_1,\ldots,a_n]$

All Other Cases:

$trans(\phi) := \phi$

Besides the definition of lower level predicates, the translated program also contains the definition of the following high-level predicates.

```
isnf([]).

l(T) :-

    append(U,[W|V],T),
    isnf(V),
    equiv([W|V],X),
    append(U,X,S),
    l(S).

v([X|T],Y) :-

    not(isnf(T)),
    append(U,[W|V],T),
    isnf(V),
    equiv([W|V],Z),
    append(U,Z,S),
    v([X|S],Y).

dequiv(X,X) :- isnf(X).

dequiv(X,Y) :-

    append(U,[W|V],X),
    isnf(V),
    equiv([W|V],Z),
    append(U,Z,Q),
    dequiv(Q,Y).

tequiv(X,Y) :-

    dequiv(X,Z),
    dequiv(Y,Z).
```

8. PROOF OF TERMINATION OF TRANSLATED PROGRAMS

In this section we define a grammar that is satisfied by all Prolog programs that result from specifications satisfying the specification grammar, and we prove that all Prolog programs satisfying this grammar terminate under certain conditions.

8.1 Definition of Prolog Grammar

The grammar for terminating Prolog programs is very similar to the grammar for trace specifications. Therefore, in order to avoid needless repetition, this section is brief and frequently refers to the section on the specification grammar.

''Occurrences'' are defined as for specification assertions. The assignment of ''in'' and ''out'' arguments for predicates follow the assignment for specification predicates and functions, with the following additions:

(1) When a predicate is created from a function by adding a new argument for the function value, that argument is designated as an ''out'' argument. For example, length(T) becomes length(T,X); the second argument is an ''out'' argument.

(2) Several new predicates appear in the translated specifications. These are assigned argument types as shown in Table 2.

Table 2 — Assignment of Argument Types to Predicates

predicate	argument types	"in" or "out"
l(ϕ)	ϕ of type "trace"	ϕ "in"
dequiv(ϕ,τ)	both of type "trace"	ϕ "in," τ "out"
tequiv(ϕ,τ)	both of type "trace"	both "in"
listappend(ϕ,τ)	1st argument "list" 2nd argument "trace"	1st "in," second "out" OR 2nd "in," 1st "out"
ϕ is τ τ arithmetic expression	both "integer" or "real"	1st "out," 2nd "in"

An argument of type "list" is a list of variables and procedure calls.

The rule for choosing which argument of "listappend" is of type "in" is as follows. The second argument of "listappend" in a translated specification is always a variable that appears in exactly one other place in the clause. If the other occurrence of that variable is as an "in" argument of a condition or an "out" argument of the conclusion, then the type of the second argument is "out" and the first argument is "in." If the other occurrence is as an "out" argument in a condition or an "in" argument of the conclusion, then the type of the second argument is "in" and the first argument is "out." For example, in

```
equiv(A1,A2) :-

    listappend([[[front]],T,[[push,X]]],A1),
    listappend([T,[[push,X]]],A3),
    isnf(A3),
    listappend([[[push,X]],T],A2).
```

The type of the first argument of the first occurrence of "listappend" is "out," the type of the first argument of the second occurrence of "listappend" is "in," and the type of the first argument of the third occurrence of "listappend" is "in."

The predicates are assigned *levels* as follows:

Level 0: length, count, all, >, <, >=, <=, = listappend
Level 1: isnf
Level 2: equiv
Level 3: dequiv
Level 4: tequiv
Level 5: v
Level 6: l

Definition 8.1: Let L be a list of variables and traces. We define concat(L) to be the trace obtained by concatenating the members of L in reverse order.* If A is a set of lists, concat(A) is defined to be {concat(L) | L $\in$ A}.

*Recall that the order of the procedure calls appearing in a trace expression is reversed in the translated Prolog program.

The semantics of the predicates are as follows.

(1) l(X) is true if X is equivalent to a normal form trace.

(2) dequiv(X,Y) returns the normal form trace Y to which X is equivalent.

(3) tequiv(X,Y) holds if and only if X and Y are equivalent (that is, if they reduce to the same normal form trace).

(4) If L is the "in" argument, listappend(L,X) returns concat(L). If X is the "in" argument, listappend(L,X) returns a list L′ such that there is a substitution for the variables in L making L=L′ and concat(L′) = X. When queried repeatedly, listappend(L,X) returns all such L′.

(5) X is Y sets X equal to the number obtained by computing the arithmetic expression Y if X is a variable. If X is a number, it holds if and only if X is equal to the number obtained by computing Y.

We define *simple* clauses in a manner analogous to the definition of simple assertion; a clause is simple if the only logical connective used in its condition is the "," (or "and") connective. We now define variable and list admissibility in a manner analogous to the definition of variable and trace admissibility for specifications.

Definition 8.2: We say that a simple clause is variable admissible if

(1) Whenever there is an occurrence of a variable in an "out" argument of a goal of the clause, there is an occurrence of that same variable as either an "in" argument of the goal or as an "out" argument of a subgoal.

(2) Whenever there is an occurrence ω of a variable as an "in" argument of a subgoal, there is either an occurrence of that same variable in an "in" argument of the goal, or an occurrence preceding ω of that variable in an "out" argument of a subgoal.

As before, the definition of list admissibility is a little more complicated.

Definition 8.3: Let ω be an occurrence in a clause. We define $G(\omega)$ to be the graph with node set $N(\omega)$ the set of all lists L and all variables occuring as trace arguments, and edge set the set of all pairs (L_1, X) such that the subgoal listappend(L_1, X) precedes ω and of all pairs (X,Y) such that X = Y precedes ω. If L is a list (or X is an argument of type "trace"), we define $G(\omega,L)$ to be the connected component of $G(\omega)$ containing L (similarly for $G(\omega,X)$). It has node set $N(\omega,L)$ (or $N(\omega,X)$).

If $A = R \cup T$, where R is a set of list arguments and T is a set of trace arguments, we define concat(A) to be $\{\text{concat}(L) \mid L \in R\} \cup T$. If A and B are two such sets, and $<$ is a trace ordering, we say that $A <^* B$ ($A \leq^* B$) if $\text{concat}(A) <^* \text{concat}(B)$ ($\text{concat}(A) \leq^* \text{concat}(B)$).

Definition 8.4: Let $<$ be a trace ordering. A simple clause is list admissible with respect to $<$ if

(1) whenever the conclusion is of the form equiv(Y,X), then $N(\omega,Y) \; ^*\!> N(\omega,X)$, where ω denotes the end of the clause, and,

(2) for each occurrence ω of a term X appearing as an "in" argument of type "list" of a predicate of the same level as the conclusion, there is an "in" argument Y of the conclusion so that $N(\omega,Y) \; ^*\!> N(\omega,X)$.

Definition 8.5: Let < be a trace ordering. Then a clause is admissible with respect to < if it is list admissible with respect to < and it is variable admissible.

Definition 8.6: Let < be a trace ordering. Let A, B, and C be conditions, and let D be a conclusion. We say that

(1) D :- A, not(B), C. is admissible with respect to < if and only if D :- A, C. and D :- A, B, C. are both admissible with respect to <.

(2) D :- A, (B | C), E. is admissible with respect to < if and only if both D :- A, B, E. and D :- A, C, E. are admissible with respect to <.

Theorem 8.7: The translation algorithm translates admissible specification assertions into admissible Prolog clauses.

Proof: We first show that admissible simple assertions are translated into admissible simple clauses.

(1) Variable admissibility is preserved.

We want to show that the following conditions hold:

[a] If there is an occurrence of a variable in an "out" argument of a goal of the clause, then there is an occurrence of that same variable as either an "in" argument of the goal or as an "out" argument of a subgoal.

[b] If there is an occurrence ω of a variable in an "in" argument of a subgoal, then there is either an occurrence of that same variable in an "in" argument of the goal, or an occurrence preceding ω of that variable in an "out" argument of a subgoal.

The result may be concluded from the following observations about properties of translated specifications, which can be proved by inspection of the translation algorithm.

Let ω be an occurrence of a variable X in an "in" argument of a predicate in the condition. One of two things can occur. Either X occurs in the original assertion (before translation), or X was created by the translation algorithm.

If X was created by the translation algorithm, then we only have to consider the following cases.

(i) X appears in trans(pred), where pred is a predicate appearing in the condition. In this case, X appears exactly twice, once as an "in" argument, and once as an "out" argument, and the occurrence as an "out" argument precedes the occurrence as a "in" argument.

(ii) X appears as an "in" argument of "listappend" in the condition, where "listappend" does not occur inside any trans(pred). The only circumstance under which this occurs is if X also occurs as an "in" argument of the conclusion.

(iii) X appears as an "out" argument of the conclusion. In this case X also occurs as an "out" argument of a "listappend" occurring in the condition.

Now consider the case in which X was *not* created by the translation algorithm. In this case, we need to examine the following possibilities.

(i) X appears in "in" argument of a predicate in trans(pred), where pred is a predicate in the condition of the original assertion. In this case, either X appears in an "in" argument of the conclusion of the original assertion or X appears in an "out" argument of pred1, where pred1 precedes pred in the condition. In the first case X appears either in an "in" argument in the translated conclusion or in an "out" argument of a listappend preceding all occurrences of translated predicates in the condition. In the second case, X appears in an "out" argument in trans(pred1), and trans(pred1) precedes trans(pred).

(ii) X appears in an "in" argument of a listappend following all occurrences of trans(pred) in the condition. In this case, X appears in an "out" argument of the conclusion in the original assertion. Thus X appears either in an "in" argument of the conclusion or in an "out" argument of some predicate in the condition, and the same kind of arguments as in (i) may be used.

(iii) X appears in an "out" argument of the conclusion. In this case the proof is similar to that of (i) and (ii).

(2) Assertions that are trace admissible with respect to a trace order $<$ are translated into clauses that are list admissible with respect to $<$.

We need to show that two conditions hold:

[a] If the conclusion is of the form equiv(Y,X), then $N(\omega,Y) \ast> N(\omega,X)$, where ω denotes the end of the clause; and

[b] for each occurrence ω of a term X appearing as an "in" argument of type "list" of a predicate of the same level as the conclusion, there is an "in" argument Y of the conclusion so that $N(\omega,Y) \ast> N(\omega,X)$.

We will verify that [b] holds; the proof of [a] is similar.

Suppose that P is a predicate of the same level as the conclusion G. Then P appears in trans(pred), where pred is of the same level as the conclusion of the original assertion. Suppose that X appears as an "in" argument of P of type "trace." Let σ denote the occurrence of X in P. We wish to show that there is an "in" argument Y of G so that $\text{concat}(N(\sigma,Y)) \ast> \text{concat}(N(\sigma,X))$.

By the definition of the translation algorithm, there is an occurrence of listappend(L,X) in trans(pred) preceding P so that concat(L) is an "in" argument of pred. Let ω be the occurrence of concat(L) in the original assertion. Then, by the definition of admissibility, there is a trace expression T appearing as an "in" argument of the goal so that $N(\omega,T) \ast> N(\omega,\text{concat}(L))$. Again, by the definition of the translation algorithm, there is an occurrence of listappend(L′,Y) preceding the occurrence of trans(pred) so the $\text{concat}(L') = T$ and Y is an "in" argument of G of type "trace." Thus it is enough to show that $N(\omega,T)$ is a subset of $\text{concat}(N(\sigma,Y))$ and that $N(\omega,\text{concat}(L))$ is a subset of $\text{concat}(N(\sigma,X))$. We will prove the second assertion; the proof of the first is similar.

The argument will be by induction on the distance of an element of $N(\omega,\text{concat}(L))$ from concat(L) in the graph $G(\omega,\text{concat}(L))$. Clearly $\text{concat}(L) \in \text{concat}(N(\sigma,X))$; thus the result holds for distance 0. Suppose that the result holds for all members of $N(\omega,\text{concat}(L))$ of distance $< t$ from concat (L), and the the distance of S from L is t. Then there is an occurrence of $U === S$ or $S === U$ preceding ω such that the distance of U from T is $t - 1$. Thus, by the induction hypothesis $U \in \text{concat}(\sigma,L)$. Since $\text{trans}(U === S)$ is one of

listappend(L_S,A_2), A_1 = A_2, listappend(L_U,A_1),
listappend(L_S,A_2), U = A_2,
A_1 = S, listappend(L_U,A_1), or
U = S

where concat(L_U) = U and concat(L_S) = S, and trans(U === S) precedes σ, the result follows.

Finally, to show that all admissible assertions are translated into admissible simple clauses, we note that

(1) trans(not(A)) = not(trans(A))

(2) trans(A | B) = trans(A) ; trans(B), and;

(3) trans(if A then B) = not(trans(A),not(B)).

Thus the definition of admissible assertions in terms of admissible simple assertions is mapped precisely to the definition of admissible clauses in terms of admissible simple clauses.

8.2 Termination Proof

To prove termination of translated programs, we use the concept of the *search tree*. Essentially, a search tree is a record of all possible attempts to satisfy a goal, including all failures and successes. The definitions we use are essentially those from Lloyd [9], with some modifications (which are noted).

Definition 8.8: A term is defined as follows:

(1) A variable is a term.

(2) A constant is a term.

(3) If f is a function, then and $t_1,\ldots,t_n$ are terms, then $f(t_1,\ldots,t_n)$ is a term.

A *literal* is a term of the form $f(t_1,\ldots,t_n)$ (a positive literal) or the negation of such a term (a negative literal). An *expression* is obtained by joining two or more literals together by the logical connectives "and," "or," and "not."

Definition 8.9: Let $V = \{v_1,\ldots,v_n\}$ be a set of distinct variables. A substitution τ is a function from V to a set of terms. If E is an expression containing the variables $\{v_1,\ldots,v_n\}$, the expression $E\tau$ is the one obtained by performing the substitution τ on the variables of E.

If σ is a substitution so that $v_i\sigma = e_i$, we denote this by $\sigma = \{v_1/e_1,\ldots,v_n/e_n\}$.

Definition 8.10: Let A and B be two expressions, and let τ be a substitution. We say that τ is a *unifier* for A and B if $A\tau = B\tau$. We say that τ is a *most general unifier* (mgu) if for each unifier π of A and B, there is a substitution σ such that $\pi = \tau\sigma$.

In other words, τ is an mgu if it is in some sense a "first" unifier of A and B.

Example: If A = f(X,3) and B =f(W,Z), the substitutions $\tau = \{X/P, W/P, Z/3\}$, and $\sigma = \{X/7, W/7, Z/3\}$ are both unifiers, but only τ is an mgu.

Note that one mgu can be obtained from another by a simple renaming of variables. Thus we can think of all mgu's as being equivalent, and so any two expressions that can be unified have a unique mgu. From now on, instead of saying "τ is *an* mgu for A and B," we will say "τ is *the* mgu for A and B."

Definition 8.11: A *program clause* is a clause of the form A :- E, where A is a positive literal and E is a (possibly empty) expression. A set of program clauses is called a *program*.

We are now ready to define a search tree. We begin with the definition of search tree for programs defined by clauses of the form

$$A :- B_1,\ldots,B_n,$$

where A is a positive literal, and the right-hand side is either empty or consists of positive literals.

Definition 8.12: Let P be a program defined by clauses of the form

$$A :- B_1,\ldots,B_n,$$

where A is a positive literal and the right-hand side is either empty or all the B_i's are positive literals. Let G be a goal of the form $C_1,\ldots,C_k$, where the C_i's are positive literals. The Prolog search tree for G is defined as follows:

(1) Each node of the tree is a goal (possibly empty).

(2) The root node is G.

(3) Let $A_1,\ldots,A_n$ be a node in the tree. This node has a descendent for each clause A :- $B_1,\ldots,B_k$, where A is unifiable with A_1. This descendent is of the form $(B_1,\ldots,B_k,A_2,\ldots,A_n)\tau$, where τ is the mgu for A_1 and A.

(4) If a node is empty, it has no descendents.

We call empty nodes *success nodes*.

The above definition is a special case of the definition of SLD-tree using rule R given in Lloyd, with the computation rule R used being the Prolog rule of always chosing the leftmost goal in a node.

Definition 8.13: Let N be a node in a search tree. If N has no ancestors other than the original goal, then the *computed substitution* at N is defined to be the mgu used to obtain N. Otherwise the computed substitution at N is defined to be $\tau\sigma$, where τ is the computed substitution at N's parent, and σ is the mgu used to obtain N. If N is a success node, the computed substitution at N is called an *answer substitution*.

A Prolog derivation of G terminates if the search tree for G is finite. Moreover, if σ is the substitution at the leftmost success node, that is the substitution returned by the program.

It is *not* true that if the Prolog derivation of G terminates, then the corresponding search tree is finite. All that is required is that the part of the tree to the left of the first success node be finite. However, the finiteness of the search tree can be used to show that other Prolog derivations terminate. Suppose, for example, we know that the search tree of a goal G is finite, and we are attempting to derive the goal (G,H). Prolog finds the substitution σ at the leftmost success node of G and attempts to derive $H\sigma$. If that fails, it tries again with $H\tau$, where τ is the substitution at the next success node of G and so on, until it either succeeds or has searched the entire tree. For this reason the search tree is a valuable tool for proving termination of Prolog programs by induction.

Let us now extend the concept of search tree to include goals that may be more general expressions. The definition we use is essentially the definition of SLDNF-tree given in Lloyd with some slight modifications for our own purposes.

Definition 8.14: Let P be a program and let G be an expression. The *search tree family of G* is defined as follows. We construct a "father tree" with the following properties:

(1) Each node of the tree is a goal.

(2) The root node is G.

(3) If $A_1,\ldots,A_k$ is a node in the tree then one of the following occurs:

[a] whenever A_1 is a positive literal, then the children of the node are calculated as in the definition of search tree;

[b] whenever $A_1 = (B \mid C)$, then the children of $A_1,\ldots,A_k$ are the children of $B,A_2,\ldots,A_k$, followed by the children of $C,A_2,\ldots,A_k$, and;

[c] whenever $A_1 = \text{not}(B)$ then if B has a finite search tree with no success nodes, the child of $A_1,\ldots,A_k$ is $A_2,\ldots,A_k$. Otherwise $A_1,\ldots,A_k$ is a failure node.

The search tree family of G is defined to be the union of the singleton set containing the "father tree" with the set of search tree families of all goals B such that $\text{not}(B),A_1,\ldots,A_k$ is a node in the "father tree."

We say that the search tree family of G is finite if it consists of a finite number of search trees, all of which are finite.

Lemma 8.15: Let A, B, C, and D be goals. Then

(1) (A, (B|C), D) has a finite search tree family if and only if (A, B, D) and (A, C, D) have finite search tree families, and;

(2) (A, not(B), C) has a finite search tree family if (A, C) and (A, B, C) have finite search tree families.

Moreover, σ is a substitution at a success node of (A,B,D) or (A,C,D) if and only if it is a substitution at a success node of (A,(B|C),D). If σ is a substitution at a success node of (A,C), then it is a substitution of a success node of (A, not(B), C).

Proof: In the first case, the father search tree of (A, B, D), (A, C, D), and (A, (B | C), D) are isomorphic to the search trees obtained by taking the father search tree for A and appending to each success node with substitution σ the father search trees for $(B,D)\sigma$, $(C,D)\sigma$ and the trees for both $(B,D)\sigma$ and $(C,D)\sigma$, respectively. The remaining search trees in the family are obtained by taking the union of the search tree families of (A, B, D) and (A, C, D), minus the parent search trees. Thus both results follow.

In the negation case, the father search tree of (A, not(B), C) is isomorphic to the tree obtained by taking the father search tree of A and appending to each success node with substitution σ such that $B\sigma$ fails the father search tree of $C\sigma$. This is isomorphic to a subtree of the father search tree of (A,C), and each substitution τ at a success node of this tree is also a substitution at a success node of (A,C). The remaining search trees in the family are a subset of the union of the search tree family of each $B\sigma$ such that $(B, C)\sigma$ is a node of (A, B, C) and the search tree family of (A, C).

The proof of termination is by induction on a partial order on goals defined as follows:

Definition 8.16: Let G and H be positive literals. Let $<$ be a trace order. We say that $G < H$ if level(G) $<$ level (H) or level(G) = level (H), all "in" arguments of both H and G are ground, and for each "in" argument T of G of type "trace," there is an "in" argument S of H of type "trace" such that $S > T$.

Lemma 8.17: Let G :- $A_1,\ldots,A_k$ be an admissible simple clause with respect to a trace order $<$. Let τ be a substitution such that all "in" arguments of $G\tau$ are ground and $(A_1,\ldots,A_t)\tau$ is ground. Then all "in" arguments of $A_{t+1}\tau$ are ground, and $A_{t+1}\tau < G\,\tau$. Moreover, if t = k, then $G\tau$ is ground, and, if G = equiv(Y,X), then $Y\tau > X\tau$.

Proof: That all "in" arguments of $A_{t+1}\tau$ are ground follows directly from the definition of variable admissibility. Thus we only need to prove that $A_{t+1}\tau < G\tau$. If level(A_{t+1}) $<$ level(G), the result holds by definition. Suppose that level(A_{t+1}) = level(G).

Let ω be the occurrence of an "in" argument X in A_{t+1}. By the definition of list admissibility, there is an "in" argument Y of G so that $N(\omega,Y) \mathbin{*>} N(\omega,X)$. By definition of the predicates "listappend" and "equal," and the fact that τ makes all members of $N(\omega,Y)$ and $N(\omega,X)$ ground, the substitution τ sets all members of $\text{concat}(N(\omega,Y))\tau$ equal to each other and to $Y\tau$, and all members of $\text{concat}(N(\omega,X))\tau$ equal to each other and to $X\tau$. Since there is a list L in $N(\omega,Y)$ and a list L′ in $N(\omega,X)$ such that concat(L) $>$ concat(L′), we have $Y\tau = \text{concat}(L\tau) > \text{concat}(L'\tau) = X\tau$, and thus $Y\tau > X\tau$.

The proof of the second part of the lemma is similar.

We are now ready to prove termination properties of translated specifications. We will not attempt to prove any properties of proofs of goals of level 0, since many of them are system predicates, and the remaining level 0 goals are not defined in terms of the rules of the grammar. We will, however, make the following assumptions about goals of level 0:

> Let G be a predicate of level 0, and let σ be a substitution such that all "in" arguments of $G\sigma$ are ground. Then $G\sigma$ has a finite proof tree family, and $G\sigma\tau$ is ground for each answer substitution τ.

Theorem 8.18: Let S be a specification admissible with respect to a trace order $<$, and let **P** be its translation. Suppose that, whenever G is a goal of level 0 with all "in" arguments ground, then G has a finite search tree family such that $G\sigma$ is ground for each answer substitution σ. Then, if G is a goal of level > 0 with all "in" arguments ground,

(1) G has a finite search tree family;

(2) $G\sigma$ is ground for each answer substitution σ, and;

(3) If G = equiv(X,Y), and ρ is an answer substitution, $X\rho > Y\rho$.

Proof: The proof is by induction on the partial order on goals. Suppose that G is a goal of level $\geq$ 0, that all "in" arguments of G are ground, and that (1) and (2) hold for all goals less than G and (3) holds for all "equiv" goals less than G.

We now consider the search tree family of G. Every child node of G is obtained by the unification of G with a goal H of some clause H :- K in **P**. Let τ be the substitution achieving this unification. Then all "in" arguments of $H\tau$ are ground, and we wish to prove that $K\tau$ has a finite search tree family.

We have two cases to consider, the case in which the clause is admissible, and the case in which the clause is one defining "l," "v," "tequiv," or "dequiv" in the Prolog utility code.

We consider the admissible case first. Suppose that H :- K is not simple. Then either K = A, not(B), C or K = A, (B | C), D. By the definition of admissibility, in the first case H :- A, C and H :- A, B, C are admissible clauses, and in the second case H :- A, B, D and H :- A, C, D are admissible clauses. Moreover, by Lemma 8.15 it is enough to show the proof tree family of $(A, C)\tau$ and $(A, B, C)\tau$ are finite in order to show that the proof tree family of $(A, \text{not}(B), C)\tau$ is finite, and it is enough to show that the proof tree families of $(A, B, D)\tau$ and $(A, C, D)\tau$ are finite in order to show that the proof tree family of $(A, (B \mid C), D)\tau$ is finite. By repeated application of Lemma 8.15, we eventually can reduce the problem to proving the finiteness of the proof tree families of a set of goals $K_1\tau$ through $K_n\tau$, where each K_i consists of positive literals and the clauses H :- K_i are admissible. In other words, we are reduced to proving the assertion in the case that H :- K is a simple admissible clause. We prove the result by showing that $K = (A_1,\ldots,A_k)$ becomes ground at each success node, and then using Lemma 8.17.

The proof is by induction on r, for $1 \leq r \leq k$. For each such r, one of two things can occur. Either the search tree of $(A_1,\ldots,A_r)\tau$ has no success nodes, in which case the search tree of $(A_1,\ldots,A_k)\tau$ is the search tree of $(A_1,\ldots,A_r)\tau$, or the search tree of $(A_1,\ldots,A_k)\tau$ is the search tree of $(A_1,\ldots,A_r)\tau$, with the search tree of $(A_{r+1},\ldots,A_t)\mu$ appended to each success node, where μ is the substitution at that node.

By Lemma 8.17, all "in" arguments of A_l are ground, and $A_l\tau < H\tau = G\tau$. Since the definition of the partial order on goals depends only upon the level of the goals and the ordering of the "in" arguments, and the substitution τ does not change the ground "in" arguments of G, we have $A_l\tau < G$. Thus, by the induction hypothesis, $A_l\tau$ has a finite search tree, with all "in" arguments ground at each success node.

Suppose that we have shown that $(A_1,\ldots,A_r)\tau$ has a finite search tree, and at each success node, $(A_1,\ldots,A_r)\mu$ is ground. We wish to show that $A_{r+1}\mu$ has a finite search tree and that at each success node, $(A_1,\ldots,A_{r+1})\rho$ is ground, where ρ is the computed substitution of that node. Since $(A_1,\ldots,A_r)\mu$ is ground, we have $(A_1,\ldots,A_r)\mu = (A_1,\ldots,A_r)\rho$. Thus all that remains to be shown is that

(1) $A_{r+1}\mu$ has a finite search tree;

(2) $A_{r+1}\rho$ is ground.

By Lemma 8.17, all "in" arguments of $A_{r+1}\rho$ are ground and $A_{r+1}\rho < G\rho$, and hence $A_{r+1}\rho < G$. Thus, by the induction hypothesis on goals, $A_{r+1}\mu$ has a finite search tree, and all arguments of $A_{r+1}\rho$ are ground.

We have now shown, by induction on r, that the search tree of $(A_1,\ldots,A_k)\tau$ is finite and that $(A_1,\ldots,A_k)\rho$ is ground at each success node (where ρ is the substitution at that node). We can thus conclude, by Lemma 8.17, that G is ground at each success node, and if μ is a substitution at a success node, and G = equiv(X,Y), then $X\mu > Y\mu$.

The case in which H :- $A_1,\ldots,A_k$ is not admissible remains to be proved. We will prove the result for the clause defining legality; the other clauses are essentially the same.

The clause defining legality is given as follows:

```
l(T) :-

    append(U,[W|V],T),
    isnf(V),
    equiv([W|V],X),
    append(U,X,S),
    l(S).
```

If we keep in mind that "append(A,B,C)" is merely the definition of "listappend([A,B],C)," we see that the clause

```
l(T) :-

    append(U,[W|V],T),
    isnf(V),
    equiv([W|V],X),
    append(U,X,S).
```

is admissible. Thus, by the reasoning given by the proof of termination in the admissible case, we have that

```
    (append(U,[W|V],T),
    isnf(V),
    equiv([W|V],X),
    append(U,X,S))τ
```

has a finite proof tree such that all arguments are fully instantiated at each success node, and if π is the substitution given at a particular success node, that concat(X)π < concat([W|V])π. Hence by Condition 3 on the definition of trace order (Definition 6.4), we have concat([U,X])π < concat ([U,[W|V]])π. (This is the only part of the termination proof in which Condition 3 is used.) Since $S\tau$ = concat([U,X])π and $T\pi$ = concat([U,[W|V]])π, we have $S\tau < T\pi$. Thus l(S)π < l(T)τ, and by the induction hypothesis l(S)π has a finite search tree.

The proofs for "v," "dequiv," and "tequiv" are similar, and we omit them.

9. SOUNDNESS OF NEGATION

We have characterized a set of trace assertions that can be translated into terminating Prolog programs, and we have provided a proof that such programs terminate, given the appropriate input. However, we have not yet guaranteed the soundness of such programs against problems that can arise from the definition of negation used in Prolog.

As an example of what can go wrong, consider the following from Lloyd [9]:

```
q(a).
p(b).

?- not(q(X)), p(X).
?- p(X), not(q(X)).
```

Prolog will return "no" to the first question, and "yes" to the second one. This is because, in order to answer the first question, Prolog attempts to verify that there exists no X such that q(X) holds, and fails, while, in order to answer the second question, Prolog finds an X such that p(X) holds (namely X = b) and then verifies that q(b) does not hold. Thus in the first case, the question is "is there no X such that q(X) holds, and is there an X such that p(X) holds," while in the second case, the question is "is there an X such that p(X) holds and q(X) does not hold."

One solution to this problem is to insist that all negated literals be ground at the time they are being proved. In Ref. 9 it is shown that a computation rule that guarantees this property is sound with respect to negation. However, a requirement may prove to be too restrictive. At times a user may want to ask such a question such as "is it true that there is no X such that q(X) holds?" We note however, that the query "not(q(X)), p(X)" is semantically identical to the query "not(q(Z)), p(X)" as far as Prolog is concerned, and the meaning of the second query is clear to the user. Thus it makes sense to relax the restriction to the requirement that, if the program attempts to prove "not(B)" at a given node in a proof tree, and some variables appearing in "not(B)" are not instantiated to ground at that point, then those variables do not appear in any descendants of that node.

Such a property can be guaranteed with respect to the Prolog computation rules by the use of assertions that satisfy the following requirement, which we call "negation admissibility."

Definition 9.1: A negation trace assertion is negation admissible if one of the following holds:

(1) The assertion is simple.

(2) The assertion is of the form A & (B | C) & D -> E, and A & B & D -> E and A & C & D -> E are both negation admissible.

(3) The assertion is of the form A & not(B) & C -> E, where

[a] whenever a variable appears in B, then it either appears in A or in an "in" argument of E, or it does not appear in C; and

[b] A & C -> E and A & B &C -> E are both negation admissible.

(4) The assertion is of the form A & (if B then C) & D -> E, where

[a] whenever a variable appears in B or C, then it either appears in A or in an "in" argument of E, or it does not appear in D; and

[b] A & D -> E, A & B & D -> E, and A & B & C & D -> E are all negation admissible.

One can then verify that, if a variable appears inside a negation, either it is ground when it is the term in which it appears inside the negation is being proved, or it is never used again after of the negation.

Negation admissible specifications, while logically sound, may still be open to misinterpretation by the user, since it is not made explicit which variables are to be universally quantified, and which are to be ground at the time they are called. However, it is a simple matter to allow the specification writer to declare explicitly which variables are to be universally quantified.

10. CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

In this report we have presented the problem of writing trace specifications so that they can be implemented correctly as Prolog programs. We have presented a grammar for trace specifications that can be executed as terminating Prolog programs, an algorithm for interpreting such specifications in Prolog, and a proof that the resulting programs terminate. We have also indicated how the problems arising from the unsoundness of the Prolog definition of negation can be avoided.

So far, the translator has been used on relatively small-sized specifications, consisting of about 25 clauses at most. Experience with some larger specifications of real-life systems suggests that several modifications need to be made to the translator before it can be scaled up. One of these is to allow specifications to use other, lower level specifications. For example, one could define a normal form trace for a binary tree as a trace that satisfies the definition of normal form for a tree and also has some other properties. One could also use such a capability to allow the user to define his or her own trace predicates. Such user-defined predicates, for example, turned out to be very useful in a specification of a multilevel secure message system [10].

Another hurdle that the translator must overcome before being usable for large-scale specifications is the complexity of the definition of admissibility. Although admissibility is relatively easy to verify for short clauses, more complex clauses containing many instances of nested nots, ors, and if-thens could be hard to check. One method of making admissibility easier to handle would be to introduce software to help the user keep track of the variables and arguments that must be considered at a particular point in the specification, possibly incorporating it in an interactive text editor. We note here that a straightforward implementation of the definition of admissibility would probably not be practical, since it would result in an algorithm whose run-time would be exponential in the number of nots, ors, and if-thens. Thus it would be necessary to develop more efficient algorithms for keeping track of the variables and arguments necessary to guarantee admissibility.

A third area of concern lies in the efficiency of translated programs. In the interest of ease of verifiability, we have up to this point avoided using the various control features of Prolog (in particular "cut") that have the potential of introducing unsoundness. However, lack of control features could make the translated specifications unacceptably slow. Thus it is necessary to investigate logically sound methods of introducing control features into translated specifications. This in turn creates a need for a more rigorous definition of the semantics of the translated programs than the rather informal one we have given in this report.

Another possible extension of the program, unrelated to the problem of upscaling, is to give it the capability of verifying consistency of specifications. A specification can be inconsistent in one of three possible ways: it can specify a trace as being both legal and illegal, it can specify a trace as being equivalent to more than one normal form trace, or it can assign to a trace two different values. The restriction to Horn clauses in Prolog makes the first kind of inconsistency impossible; a trace is illegal if and only if it satisfies no definition of legality. The presence or absence of the second and third kinds of inconsistency can be verified for individual traces by using the system as is described in this report; one simply uses "v(T,X)" or "dequiv(T,S)" followed by the ";" command. But the system does not tell the user when a given specification is consistent, that is, when no traces with two different values exist. Thus one further extension to the system could be a means of analyzing the definitions of value and equivalence given in a specification to determine whether or not they give rise to inconsistencies.

11. ACKNOWLEDGMENTS

I thank Thor Bestul for his comments and John McLean and David Mutchler for their comments on an earlier version of this report.

12. REFERENCES

1. A.W. Bartussek and D.L. Parnas, "Using Traces to Write Abstract Specifications for Software Modules," UNC Rep. TR 77-012, University of North Carolina, Chapel Hill, NC, 1977.

2. J.D. McLean, "A Formal Method for the Abstract Specification of Software," *J. ACM* **31**, 600-627 (1984).

3. J.K. Dixon, J.D. McLean, and D.L. Parnas, "Rapid Prototyping by Means of Abstract Module Specifications Written as Trace Axioms," *ACM SIGSOFT Eng. Notes* **7**, 45-49 (1982).

4. J.D. McLean, D.M. Weiss, and C.E. Landwehr, "Executing Trace Specifications Using Prolog," NRL Report 8940, 1986.

5. W.F. Clocksin and C.S. Mellish, *Programming in Prolog* (Springer-Verlag, New York, 1981).

6. D.M. Hoffman, *Trace Specification of Communication Protocols*, thesis, University of North Carolina, Chapel Hill, NC, 1984.

7. D.M. Hoffman and R. Snodgrass, "Trace Specifications: Methodology and Models," Technical report, 1985.

8. R.E. Griswold and M.T. Griswold, *The Icon Programming Language* (Prentice-Hall, Englewood Cliffs, NJ, 1983).

9. J.W. Lloyd, *Foundations of Logic Programming* (Springer-Verlag, New York, 1984).

10. C. Cross, *A Trace Specification of the MMS Security Model*, manuscript, 1987, to be published as an NRL Report.